Abstraction over Public Interfaces

Dilian Gurov, Marieke Huisman

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Abstraction over Public Interfaces

Dilian Gurov — Marieke Huisman

N° 5330

October 18, 2004

Thème SYM

ISSN 0249-6399

Abstraction over Public Interfaces

Dilian Gurov , Marieke Huisman

Thème SYM — Systèmes symboliques
Projet Everest, INRIA Sophia Antipolis &
Royal Institute of Technology, Kista, Sweden

Rapport de recherche n° 5330 — October 18, 2004 — 22 pages

Abstract: The use of public interfaces as a means of encapsulating method implementations has become standard in software design, but still requires the development of appropriate verification techniques. In this paper we study the consequences on verification of dividing methods into public and private ones. We introduce a notion of *interface behaviour* of applets which abstracts from the internal, private behaviour, and we propose an abstraction technique, based on inlining of private methods, to verify properties of this interface behaviour. We define the *inlining transformation* and prove that it preserves the properties expressible in our *simulation logic*. Our abstraction technique is particularly suitable for *compositional verification*, since it allows global system properties to be inferred from the interface properties of the components that are not yet available. In addition, we show on a concrete case study how the reduction in the number of methods resulting from the inlining transformation drastically improves the performance of the computationally most expensive step in the compositional verification technique previously developed by the authors.

Key-words: interface behaviour, inlining transformation, temporal logic, compositional verification

Abstraction des Interfaces Publiques

Résumé : L'utilisation des interfaces publiques pour encapsuler des implémentations de méthodes est devenue standard dans le développement des logiciels. Par contre, il n'existe pas encore de techniques de vérification appropriées. Dans cet article, nous étudions les conséquences sur la vérification d'un partage entre méthodes publiques et privées. Nous introduisons une notion de *comportement d'interface* des applets, qui abstrait le comportement privé. De plus, nous proposons une technique de vérification, basée sur l'*inlining* des méthodes privées, afin de vérifier des propriétés de ce comportement d'interface. Nous définissons une *transformation d'inlining* et nous prouvons qu'elle préserve les propriétés qui peuvent être exprimées dans notre *logique de la simulation*. En particulier, notre technique d'abstraction est utile pour faire la *vérification compositionnelle*, parce qu'elle permet d'inférer les propriétés des systèmes globals en utilisant les propriétés d'interface des composants qui ne sont pas encore disponibles. Nous montrons sur un cas d'étude concret que la réduction du nombre des méthodes à considérer améliore la performance de notre technique pour la vérification compositionnelle.

Mots-clés : comportement d'interface, transformation d'inlining, logique temporelle, vérification compositionnelle

Contents

1 Introduction

With the emergence of small and mobile personal devices, such as smart cards, security has become a major concern. Typically, such personal devices contain privacy-sensitive information, *e.g.* financial data, health care information or electronic identities. Thus, for the widespread acceptance of the use of such devices, security of such private information needs to be guaranteed.

Ideally, a smart device user should have the possibility to install new applications (usually called *applets*) by need. Therefore, efficient verification techniques are needed that enable to check whether a new applet might break the security of the smart device. In earlier work, we developed a control-flow based compositional verification technique supporting post-issuance loading of applets; and we showed feasibility of this technique by applying it to an industrial case study [5, 9]. Our technique prescribes the specification of structural local properties for the different applets, plus the specification of a structural or behavioural global property for the composed system. We check once whether the local applet properties are sufficient to guarantee the global system property; and for each newly installed applet we check (possibly on-device) that it respects its local property.

Applying our techniques to the industrial case study demonstrated a shortcoming in our approach: our model did not provide any abstraction. In particular we only considered applet interfaces containing *all* methods implemented by the applet, and we did not have a notion of public interfaces containing *only* methods that are visible to the outside world. As a consequence, the global and local properties had to take many of the implementation details into account, while it would have been more natural to describe them in terms of the public interfaces only. For example, to specify the local properties in the case study we needed to compute the set of private methods reachable from particular public methods. In fact, in a truly compositional setting one only knows the public interface of the new applet, and does not have access to any implementation details, thus properties can only be specified at the public level.

Moreover, this lack of abstraction caused a blowup in the size of the formulae and of the models used for the verifications – for both their size depends on the size of the interface considered. To give an indication of the improvements that can be expected: in the industrial case study we considered an electronic purse applet which implemented 367 methods, of which only 4 were public, all others were private. And indeed, in the original case study we were not able to complete the decomposition check, because the size of the models was too large.

To verify properties of the public interface behaviour of applets, we propose a technique to abstract away from private behaviour. The technique is based on inlining of private methods. We show the abstraction to be sound with respect to public interface properties: every property that holds of the public interface behaviour of the inlined applet also holds of the public interface behaviour of the original applet. Since for the inlined applet its public interface behaviour coincides with its behaviour (since it has no private behaviour), the inlining transformation provides a means for checking properties of the interface behaviour of the original program by reduction to a standard verification problem. Moreover, in case

the concrete implementation is last-call recursive, the abstraction technique is complete with respect to observable public interface properties; hence, if such a property does not hold of the inlined applet it does not hold of the original applet as well. We do not have completeness in general, because the abstraction transformation may introduce new observable behaviours.

Using the abstraction techniques described in this paper, our improved scenario for secure post-issuance loading now is the following.

1. Define a set of public methods M used for interaction between applets $\mathcal{A}_1, \ldots, \mathcal{A}_n$.
2. For each applet $\mathcal{A}_i$ specify a structural local property $\sigma_{\mathcal{A}_i}$ using only the methods in M.
3. Specify a public global property ϕ over M, that should hold for the composed system.
4. Compute maximal models $\mathcal{M}ax(\sigma_{\mathcal{A}_i})$ for each applet $\mathcal{A}_i$, and verify that their composition satisfies ϕ, *i.e.* $\mathcal{M}ax(\sigma_{\mathcal{A}_1}) \uplus \ldots \uplus \mathcal{M}ax(\sigma_{\mathcal{A}_n}) \models \phi$.
5. For each implementation of applet $\mathcal{A}_i$, compute the abstraction $\alpha_M(\mathcal{A}_i)$, and verify that $\alpha_M(\mathcal{A}_i) \models \sigma_{\mathcal{A}_i}$.

This paper shows that all verifications can be done efficiently (In particular, Section 7 reconsiders the case study [5]). Notice that this also enables a different scenario, where a new applet comes with its own local property, and the decomposition check is repeated (possibly on-device) to ensure that this local property is sufficient to ensure global system consistency.

Even though our abstraction technique is developed in the framework of a compositional verification method, it has a methodological significance in its own. Whenever one specifies properties of the behaviour of an applet, one prefers to focus on its observable, public interface behaviour, and abstract from the private, internal behaviour.

Related work The inlining procedure as described in this paper closely resembles standard inlining procedures used in compiler optimisations, see *e.g.* [6]. However, compiler optimisations *must* be behaviourally equivalent, while our verification technique only requires that all existing behaviours are preserved by inlining. We believe that our approach for proving property preservation is applicable to such compiler optimisations as well.

The approach of combining property preserving abstraction with verification is standard, see *e.g.* [3]. Usually, the goal of applying abstraction is to obtain a smaller or simply finite model for verification. In our case, the primary purpose of applying the inlining transformation is different: to reduce the problem of verifying a property of the public interface behaviour of an applet to verifying a property of the (usual) behaviour of the transformed applet.

Further we should mention the temporal logic of calls and returns CaRet [1]. This logic allows to specify properties in terms of method calls and returns. A special verification strategy is defined, that is able to jump over internal computations. Our approach is the

opposite: we compute an abstract model, and use standard verification techniques to verify properties – expressed in a standard temporal logic – on this abstract model.

Overview of the paper The remainder of this paper is organised as follows. Sections 2 and 3 introduce the necessary background, and in particular the logic and program model that we use. Section 4 defines the behaviour of an applet *w.r.t.* a set of public methods. Next, Section 5 presents the inlining algorithm that forms the basis of our abstraction technique, and proves that it is property preserving. Section 6 describes formally how the abstraction techniques are used for compositional reasoning. Finally, Section 7 revisits the industrial case study, and shows the practical impact of the abstraction techniques, while Section 8 draws more general conclusions on the applicability of our method.

2 Simulation and Logic

First, we briefly recall some definitions and results that form the basis for our compositional verification method. For a full overview, the reader is referred to [8, 9]. We use a subset of the modal μ-calculus [7] as our specification language. We exploit that formulae in this subset can be characterised by simulation, and vice versa, therefore we call this logic *simulation logic*. Throughout, we fix a set of labels L and a set of atomic propositions A.

Definition 1. [Simulation Logic] *The formulae of* simulation logic *are inductively defined by:*

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\,\phi \mid \nu X.\phi$$

where $p \in A$ and $a \in L$.

Next, we define a general notion of model and specifications.

Definition 2. [**Model**] *A* model *is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a transition relation, and $\lambda\colon S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state s the atomic propositions that hold in s. A* specification *$\mathcal{S}$ is a pair $(\mathcal{M},E)$, where $\mathcal{M}$ is a model and $E \subseteq S$ is a set of states.*

Intuitively, one can think of E as the set of entry states of the model. For specifications, we define the usual notions of satisfaction $\models$ and simulation $\leq$ (where related states satisfy the same atomic propositions). This simulation relation preserves (backwards) logical properties.

Theorem 1. *$\mathcal{S}_1 \leq \mathcal{S}_2$ and $\mathcal{S}_2 \models \phi$ implies $\mathcal{S}_1 \models \phi$*

Proof. Corollary 2.16 in [9] □

Weak simulation and logic. In Section 4 we show how private method calls can be abstracted away into internal transitions, labelled with the distinguished *silent* action ε. When abstracting in such a way from part of the concrete behaviour of a system, one also has to abstract from the internal behaviour, and instead consider the visible behaviour in terms of *weak* transitions. We use the standard definition of weak transitions $s \stackrel{a}{\Rightarrow} t$ in terms of strong transitions: $s \stackrel{\varepsilon}{\Rightarrow} t$ whenever $s(\stackrel{\varepsilon}{\rightarrow})^* t$, and $s \stackrel{a}{\Rightarrow} t$ whenever $s \stackrel{\varepsilon}{\Rightarrow} \stackrel{a}{\rightarrow} \stackrel{\varepsilon}{\Rightarrow} t$, for all $a \neq \varepsilon$. Weak simulation $\leq_w$ is then defined as simulation *w.r.t.* weak transitions. Similarly, for the weak satisfaction relation $\models_w$, we interpret the box modality over the weak transitions. As above, the weak simulation relation preserves weak satisfaction of logical properties.

Theorem 2. *$\mathcal{S}_1 \leq_w \mathcal{S}_2$ and $\mathcal{S}_2 \models_w \phi$ implies $\mathcal{S}_1 \models_w \phi$*

Proof. Immediate consequence of Theorem 5 in [8]. □

Finally, a standard transformation from weak to strong formulae exists [10]. This transformation, which we call δ, has the following property.

Proposition 1. *$\mathcal{S} \models_w \phi$ iff $\mathcal{S} \models \delta(\phi)$.*

3 Applet Structure and Behaviour

Our program model is control-flow based, and defines two different views on applets: a structural and a behavioural view. Both views are instantiations of the general notions of model and specification, allowing the results presented above to be instantiated at both levels. Notice in particular that these instantiations yield a structural and a behavioural version of simulation and simulation logic. Again, we refer to [8, 9] for more detail.

Applet Structure Since we abstract away from all data, applet structure is defined as a collection of call graphs for the methods the applet implements. Notice that since so-far smart cards are our application domain, we only have to consider sequential methods. A method specification is an instance of the general notion of specification.

Definition 3. (Method specification) *A* method graph *for $m \in \mathcal{M}eth$ over a set M of method names is a finite model*

$$\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$$

where V_m is the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, $m \in \lambda_m(v)$ for all $v \in V_m$, i.e. *each node is tagged with its method name, and the nodes $v \in V_m$ with $r \in \lambda_m(v)$ are* return points. *A* method specification *for $m \in \mathcal{M}eth$ over M is a pair $(\mathcal{M}_m, E_m)$, where $\mathcal{M}_m$ is a method graph for m over M and $E_m \subseteq V_m$ is a non–empty set of* entry points *of m.*

Simulation and satisfaction, instantiated to this particular type of models are called structural simulation $\leq_s$, and structural satisfaction $\models_s$, respectively. We will write $\lambda_{\mathsf{Meth}}(v)$ to denote the function returning the name of the method to which v belongs.

$$\text{(transfer)} \qquad \frac{m \in I^+ \qquad v \rightarrow_m v' \qquad v \models \neg r}{(v, \sigma) \xrightarrow{\varepsilon} (v', \sigma)}$$

$$\text{(call)} \qquad \frac{m_1, m_2 \in I^+ \qquad v_1 \xrightarrow{m_2}_{m_1} v_1' \qquad v_1 \models \neg r \qquad v_2 \models m_2 \qquad v_2 \in E}{(v_1, \sigma) \xrightarrow{m_1 \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma)}$$

$$\text{(return)} \qquad \frac{m_1, m_2 \in I^+ \qquad v_2 \models m_2 \wedge r \qquad v_1 \models m_1}{(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1, \sigma)}$$

Table 1: Applet Transition Rules

Interface Next we define the notion of applet interface. For each applet, we distinguish an *implementation interface,* defining all methods provided and required by the applet, and a *public interface,* defining all methods that are visible to and used from other applets. Let $\mathcal{M}eth$ be a countably infinite set of method names.

Definition 4. (Applet interface) *An* applet interface *is a pair* $I = (I^+, I^-)$*, where* $I^+, I^- \subseteq \mathcal{M}eth$ *are finite sets of names of* provided *and* required *methods, respectively. The* composition *of two interfaces* $I_1 = (I_1^+, I_1^-)$ *and* $I_2 = (I_2^+, I_2^-)$ *is defined by* $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$.

To formally define the notion *applet with implementation interface*, we use the notion of disjoint union of specifications $\mathcal{S}_1 \uplus \mathcal{S}_2$, where each state is tagged with 1 or 2, respectively, and $(s, i) \xrightarrow{a}_{\mathcal{S}_1 \uplus \mathcal{S}_2} (t, i)$, for $i \in \{1, 2\}$, if and only if $s \xrightarrow{a}_{\mathcal{S}_i} t$.

Definition 5. (Applet) *An* applet $\mathcal{A}$ *with implementation interface* I*, written* $\mathcal{A} : I$*, is defined inductively by*

- $(\mathcal{M}_m, E_m) : (\{m\}, M)$ *if* $(\mathcal{M}_m, E_m)$ *is a method specification for* $m \in \mathcal{M}eth$ *over* M*, and*
- $\mathcal{A}_1 \uplus \mathcal{A}_2 : I_1 \cup I_2$ *if* $\mathcal{A}_1 : I_1$ *and* $\mathcal{A}_2 : I_2$.

An applet is *closed* if $I^- \subseteq I^+$, *i.e.* it does not require any external methods.

The *public interface* of an applet $\mathcal{A}$: I is characterised by a set of methods M such that $M \subseteq I^+$: the set of methods publicly provided by the applet is M, while the set of publicly required methods is $I^- - (I^+ - M)$; thus applet $\mathcal{A}$: I has public interface $(M, I^- - (I^+ - M))$.

Applet Behaviour Next we instantiate specifications on the behavioural level.

Definition 6. (Behaviour) *Let $\mathcal{A} = (\mathcal{M}, E) : I$ be a closed applet where $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{A}$ is described by the specification $b(\mathcal{A}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$ such that $S_b = V \times V^*$,* i.e. *states are pairs of control points and stacks, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\varepsilon\}$, $\rightarrow_b$ is defined by the rules of Table 1, $A_b = A$, and $\lambda_b((v, \sigma)) = \lambda(v)$.*

The set of initial states E_b is defined by $E_b = E \times \{\epsilon\}$, where ϵ denotes the empty sequence over V.

Note that applet behaviour defines a push down automaton (see, *e.g.*, [2] for a survey of verification techniques for infinite process structures). We exploit this by using a model checker for PDAs to verify behavioural properties.

Also on the behavioural level, we instantiate the definitions of simulation $\leq_b$ and satisfaction $\models_b$. Any two applets that are related by structural simulation, are also related by behavioural simulation (Theorem 3.9 in [9]), but the converse is not true (since behavioural simulation only requires reachable states to be related).

For convenience, below we will often write the states of the behavioural model as a simple sequence of states, *i.e.* $v \cdot \sigma$, instead of (v, σ). We use reverse indexing to denote the i^{th} element from the back of a sequence, so that $(v \cdot \sigma)_{|\sigma|} = v$ (where $|\sigma|$ denotes the length of a sequence σ), and use $last(\sigma)$ to denote σ_0.

4 Interface Behaviour

The next section defines an inlining algorithm that transforms a concrete applet implementation into an applet that contains only method calls to public methods. We want to prove that for any closed applet, every behaviour of the concrete applet is also a behaviour of the inlined applet. However, for this to hold, we have to abstract the concrete behaviour to the level of public methods. Therefore, we introduce the notion of *interface behaviour* of an applet *w.r.t.* a set of public methods.

First we define the *top* public method *w.r.t.* M, that given a callstack σ returns the first public method to which a node in the call stack belongs.

$$\begin{aligned} \mathsf{top\_index}^M(\sigma) &= Max(\{i \mid 0 \leq i < |\sigma| \wedge \lambda_{\mathsf{Meth}}(\sigma_i) \in M\}) \\ \mathsf{top}^M(\sigma) &= \lambda_{\mathsf{Meth}}(\sigma_{\mathsf{top\_index}^M(\sigma)}) \end{aligned}$$

Using these definitions, we can define a relabelling ρ^M of transition labels to the public level. Labels for calls and returns between public methods are unchanged. A call from a private to a public method is relabelled as a call from the *top* public method in the pending call stack. A return from a public to a private method is relabelled as a return to the *top*

public method. All other transitions get labelled as silent actions.

$$\rho^M((v,\sigma),\ell) = \begin{cases} \ell & \text{if } \ell = m_1\{\mathsf{call}/\mathsf{ret}\}m_2 \wedge m_1, m_2 \in M \\ \mathsf{top}^M(v\cdot\sigma)\ \mathsf{call}\ m_2 & \text{if } \ell = m_1\ \mathsf{call}\ m_2 \wedge m_1 \notin M, m_2 \in M \\ m_1\ \mathsf{ret}\ \mathsf{top}^M(\sigma) & \text{if } \ell = m_1\ \mathsf{ret}\ m_2 \wedge m_1 \in M, m_2 \notin M \\ \varepsilon & \text{otherwise} \end{cases}$$

Now we are ready to define the interface behaviour of applet $\mathcal{A}$ *w.r.t.* a set of public methods M.

Definition 7. [Interface behaviour] *Let $\mathcal{A} : I$ be a closed applet with behaviour $b(\mathcal{A}) = ((S, L, \rightarrow, A, \lambda), E)$. Let $M \subseteq I^+$ be a set of public methods. The* interface behaviour of $\mathcal{A}$ w.r.t. M *is defined as $b^M(\mathcal{A}) = ((S, L^M, \rightarrow^M, A^M, \lambda^M), E^M)$, where*

- $L^M = \{\varepsilon\} \cup \{m_1 \ell m_2 \mid m_1, m_2 \in M \wedge \ell \in \{\mathsf{call}, \mathsf{ret}\}\}$,
- $\rightarrow^M = \{((v,\sigma), \ell, (v',\sigma')) \mid \exists a \in L.\ (v,\sigma) \xrightarrow{a} (v',\sigma') \wedge \rho^M((v,\sigma),a) = \ell\}$,
- $A^M = M \cup \{r\}$,
- $\lambda^M = (v,\sigma) \mapsto \{\mathsf{top}^M(v\cdot\sigma)\} \cup (\mathsf{if}(v \in M \wedge v \models r)\ \mathsf{then}\ \{r\}\ \mathsf{else}\ \varnothing)$,
- $E^M = \{v \mid v \in E \wedge \lambda_{\mathsf{Meth}}(v) \in M\}$.

Proposition 2. *The interface behaviour of $\mathcal{A}$* w.r.t. *I^+ is identical to its behaviour,* i.e. *$b^{I^+}(\mathcal{A}) = b(\mathcal{A})$.*

We define behavioural interface simulation $\mathcal{A} \leq_b^M \mathcal{B}$ as $b^M(\mathcal{A}) \leq b^M(\mathcal{B})$, and weak behavioural interface simulation $\mathcal{A} \leq_{b,w}^M \mathcal{B}$ as $b^M(\mathcal{A}) \leq_w b^M(\mathcal{B})$. Notice that $\mathcal{A}$ and $\mathcal{B}$ need not have the same interfaces, we only require $M \subseteq I_{\mathcal{A}}^+$ and $M \subseteq I_{\mathcal{B}}^+$. Similarly, for any formula ϕ in simulation logic over $L^M$and A^M, we define behavioural interface satisfaction $\mathcal{A} \models_b^M \phi$ as $b^M(\mathcal{A}) \models \phi$, and weak behavioural interface satisfaction $\mathcal{A} \models_{b,w}^M \phi$ as $b^M(\mathcal{A}) \models_w \phi$.

5 An Inlining Transformation

Next we define an inlining algorithm α_M that, given a set of public methods M, transforms an applet graph by inlining all private calls. Recursive calls to private methods are not inlined, but create loops in the resulting graph. We prove that the interface behaviour of the original applet $\mathcal{A}$ is simulated by the behaviour of the inlined applet $\alpha_M(\mathcal{A})$, thus (by Theorem 1) all properties ϕ of the latter, *i.e.* $\alpha_M(\mathcal{A}) \models_b \phi$, are also properties of the former, *i.e.* $\mathcal{A} \models_b^M \phi$. Moreover, we prove that if the applet is last-call recursive, the two behaviours are weak simulation equivalent – thus both applets satisfy exactly the same observable properties at the public interface level.

Notice that the inlining algorithm does not require the applet to be closed and treats all external methods as public.

The Inlining Algorithm. The algorithm inlines every public method. Intuitively, constructing the inlining for a public method m corresponds to executing the interface behaviour of m, where method calls to public methods are ignored, and recursion is eliminated and replaced by iteration. The nodes of the inlined applet can thus be seen as states of the (interface) behaviour of the original applet, modulo an abstraction function which replaces recursion by iteration.

During the inlining, each edge that represents internal transfer or a call to a public method is left unchanged. Each edge that represents a call to a private method is replaced by two internal edges: one from the calling point to the entry point of the method; and another from the return point of the method to the destination of the calling edge[1]. The private method is inlined recursively. Each node is replaced by a sequence denoting the fragment of the call stack from the activation of the public method up-to the current node (except for the case of a recursive call). Since we keep track of the pending call stack, we can recognise recursive calls to private methods. In that case, the appropriate initial fragment of the call stack is used to decide the exact new edges.

For the formal definition of the inlining algorithm, we need some new notions. Let $\mathcal{A} : I$ be an applet and $M \subseteq I^+$ be a set of public methods. An *M-frame* is a sequence of nodes σ of which only $\lambda_{\mathsf{Meth}}(\sigma_0)$ is in M. An M-frame is called *normal*, if all nodes in the frame belong to different methods. We choose to represent the nodes of the inlined applet by normal M-frames derived from the behaviour of the original applet. The abstraction function mentioned above (replacing recursion by iteration) is formalised by means of the (normalising) conditional rewrite rule $\sigma \cdot v \cdot \sigma' \cdot v' \cdot \sigma'' \hookrightarrow \sigma \cdot v \cdot \sigma''$ if $\lambda_{\mathsf{Meth}}(v) = \lambda_{\mathsf{Meth}}(v')$ and $\sigma' \cdot v' \cdot \sigma''$ is a normal M-frame. Let $\nu(\sigma)$ denote the normal form of σ *w.r.t.* the rule. Note that if σ is an M-frame, then $\nu(\sigma)$ is a normal M-frame. Moreover, for any (normal) M-frame σ we have $\mathsf{top}^M(\sigma) = \lambda_{\mathsf{Meth}}(\sigma_0)$.

Further, we define $\mathcal{I}nt$, $\mathcal{P}ub$ and $\mathcal{P}riv$, denoting the sets of internal, public and private edges of a method *w.r.t.* a set of public methods M, respectively.

$$\begin{aligned}
\mathcal{I}nt_M(m) &= \{(v, \varepsilon, v') | v \rightarrow_m v' \wedge v \models \neg r\} \\
\mathcal{P}ub_M(m) &= \{(v, m', v') \mid v \xrightarrow{m'}_m v' \wedge v \models \neg r \wedge m' \in M\} \\
\mathcal{P}riv_M(m) &= \{(v, m', v') \mid v \xrightarrow{m'}_m v' \wedge v \models \neg r \wedge m' \notin M'\}
\end{aligned}$$

The definition of the inlining algorithm uses auxiliary functions χ and ζ. The function χ considers all edges related to a method: it returns internal and public edges with renamed nodes – using the pending call stack, and calls method ζ on private edges. Function ζ adds edges to the entry point, and from the return point of the private method, using the pending call stack argument, and if necessary normalising the result (this uses the fact that the pending call stack is always a normalised M-frame). Then it checks if the private call is non-recursive, in which case the private method is inlined recursively.

[1] If a method has several entry or return points, several internal edges are created.

Definition 8. [Inlined applet] *Let $\mathcal{A} : I$ be an applet, and let M be a set of public methods, such that $M \subseteq I^+$. Let M' be the set $M \cup (I^- - I^+)$. We define the* inlined applet *$\alpha_M(\mathcal{A}) = ((V', L', \rightarrow', A', \lambda'), E')$, where*

- $V' = \{w \in V^+ \mid w \text{ is a normal } M\text{-frame}\}$,
- $L' = M' \cup \{\varepsilon\}$,
- $\rightarrow' = \bigcup_{m \in M} \gamma(m, \epsilon)$ *where*

$$\begin{array}{rcl} \chi(m,\sigma) & = & \{(v \cdot \sigma, \ell, v' \cdot \sigma) \mid (v, \ell, v') \in \mathcal{I}nt_M(m) \cup \mathcal{P}ub_M(m)\} \cup \\ & & \bigcup_{(v,m',v') \in \mathcal{P}riv(m)} \zeta(\sigma, (v, m', v')) \end{array}$$

$$\begin{array}{rcl} \zeta(\sigma,(v,m',v')) & = & \{(v \cdot \sigma, \varepsilon, \nu(e \cdot v' \cdot \sigma) \mid e \models m' \wedge e \in E\} \cup \\ & & \{(\nu(rt \cdot v' \cdot \sigma), \varepsilon, v' \cdot \sigma) \mid rt \models (m' \wedge r)\} \cup \\ & & \mathsf{if} \quad \neg\exists i.0 \leq i \leq |\sigma| \wedge (v \cdot \sigma)_i \models m' \\ & & \mathsf{then} \quad \chi(m', v' \cdot \sigma) \\ & & \mathsf{else} \quad \varnothing \end{array}$$

- $A' = M \cup \{r\}$,
- $\lambda' = \sigma \mapsto \{\lambda_{\mathsf{Meth}}(\sigma_0)\} \cup (\mathsf{if}\ (|\sigma| = 1 \wedge \sigma_0 \models r)\ \mathsf{then}\ \{r\}\ \mathsf{else}\ \varnothing)$, and
- $E' = \{v \in E \mid \lambda_{\mathsf{Meth}}(v) \in M\}$.

Example Before discussing properties of the inlining algorithm, we first show a simple example. Suppose we have an applet as depicted in the left-hand column of Figure 1. Inlining this applet with the public method set $\{m\}$ results in the applet depicted in the right-hand column of Figure 1. Notice that all internal and public edges are preserved, while private method calls are replaced by two edges: to the entry and from the return point of the called method, respectively.

Properties We state several useful properties of the inlining algorithm. First of all, the inlining algorithm computes an applet.

Proposition 3. *Let $\mathcal{A} : I$ be an applet and $M \subseteq I^+$ a set of public methods. The inlined applet $\alpha_M(\mathcal{A})$ has interface $I_{\alpha_M(\mathcal{A})} = (M, M \cup (I^- - I^+))$,* i.e. *$\alpha_M(\mathcal{A}) : (M, M \cup (I^- - I^+))$.*

By Proposition 2 we thus get:

$$b^M(\alpha_M(\mathcal{A})) = b(\alpha_M(\mathcal{A}))$$

Since the inlining transformation α_M only affects methods not in M, α_{I^+} is the identity operation.

Proposition 4. *Let $\mathcal{A} : I$ be an applet. Then $\alpha_{I^+}(\mathcal{A}) = \mathcal{A}$.*

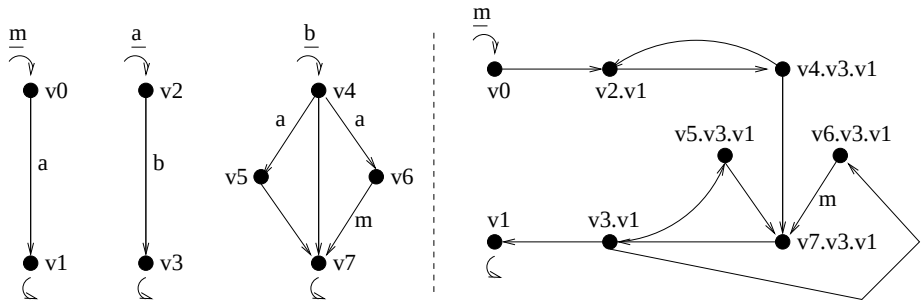

Figure 1: Example applet

Finally, the inlining algorithm enjoys the following distributivity property.

Proposition 5. *Let $\mathcal{A} : I_{\mathcal{A}}$ and $\mathcal{B} : I_{\mathcal{B}}$ be applets, and let $M_{\mathcal{A}} \subseteq I^+_{\mathcal{A}}$ and $M_{\mathcal{B}} \subseteq I^+_{\mathcal{B}}$ be such that $I^-_{\mathcal{A}} - I^+_{\mathcal{A}} \subseteq M_{\mathcal{B}}$ and $I^-_{\mathcal{B}} - I^+_{\mathcal{B}} \subseteq M_{\mathcal{A}}$. Then*

$$\alpha_{M_{\mathcal{A}} \cup M_{\mathcal{B}}}(\mathcal{A} \uplus \mathcal{B}) = \alpha_{M_{\mathcal{A}}}(\mathcal{A}) \uplus \alpha_{M_{\mathcal{B}}}(\mathcal{B})$$

Simulation Results. As already mentioned, the interface behaviour of the original applet is preserved by the inlining algorithm, *i.e.* every execution of the interface behaviour of $\mathcal{A}$ is an execution of the behaviour of $\alpha_M(\mathcal{A})$. This is due to the close correspondence between the interface behaviour of $\mathcal{A}$ and the structure of $\alpha_M(\mathcal{A})$. We provide an "inlining" transformation α'_M on the states of $b^M(\mathcal{A})$ by defining $\alpha'_M(v, \sigma) = (hd(\gamma), tl(\gamma))$, where $\gamma = \beta_M(v \cdot \sigma)$ and where $\beta_M(\sigma)$ denotes the sequence of normalised M-frames. Notice that we always have $hd(hd(\gamma)) = hd(v \cdot \sigma)$. We show that α'_M is a simulation relating the original interface behaviour with the inlined behaviour.

Theorem 3. *Let $\mathcal{A} : I$ be a closed applet, and let $M \subseteq I^+$. We have $b^M(\mathcal{A}) \leq b(\alpha_M(\mathcal{A}))$.*

Proof. We show by co-induction that α'_M is a simulation between $b^M(\mathcal{A})$ and $b(\alpha_M(\mathcal{A}))$, *i.e.*, we show that (1) the valuations of (v, σ) in $b^M(\mathcal{A})$ and $\alpha'_M(v, \sigma)$ in $b(\alpha_M(\mathcal{A}))$ agree, and (2) if $(v, \sigma) \xrightarrow{l} (v', \sigma')$ in $b^M(\mathcal{A})$, then $\alpha'_M(v, \sigma) \xrightarrow{l} \alpha'_M(v', \sigma')$ in $b(\alpha_M(\mathcal{A}))$. The result then follows since α'_M maps the entry states of $b^M(\mathcal{A})$ to entry states of $b(\alpha_M(\mathcal{A}))$ (in fact, the entry states coincide, and α'_M maps every entry state to itself).

Let (v, σ) be a configuration of $b^M(\mathcal{A})$, and hence also of $b(\mathcal{A})$. Let $\alpha'_M(v, \sigma) = (w, \gamma)$; w is a normal M-frame of $\mathcal{A}$, and thus a node of $\alpha_M(\mathcal{A})$. Note that $hd(w) = v$. It is easy to check that valuations on (v, σ) and (w, γ) agree, so we focus on the second goal. We consider

the different cases leading to transitions from configuration (v, σ) in $b(\mathcal{A})$, as induced by the transition rules for closed applets given in Table 1. Notice that in the construction of $\alpha_M(\mathcal{A})$ the auxiliary function χ initially is invoked with arguments w_0 and ϵ, and that eventually this results in a recursive call of χ with arguments $hd(w)$ and $tl(w)$.

(transfer) Let $v \rightarrow_m v'$ and $v \models \neg r$. Then $(v, \sigma) \xrightarrow{\varepsilon} (v', \sigma)$ in $b(\mathcal{A})$, and hence also in $b^M(\mathcal{A})$. Then, by definition of α_M (edge in set $\mathcal{I}nt_M(m)$) and α'_M, $w \rightarrow_{\mathsf{top}^M(w)} v' \cdot tl(w)$ is an edge in $\alpha_M(\mathcal{A})$, and $w \models \neg r$. Therefore $(w, \gamma) \xrightarrow{\varepsilon} (v' \cdot tl(w), \gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M(v', \sigma) = (v' \cdot tl(w), \gamma)$, and hence $\alpha'_M(v, \sigma) \xrightarrow{\varepsilon} \alpha'_M(v', \sigma)$.

(call) Let $v \xrightarrow{m_2}_{m_1} v'$, $v \models \neg r$, $v_2 \models m_2$ and $v_2 \in E$. Then $(v, \sigma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v' \cdot \sigma)$ in $b(\mathcal{A})$. We consider three cases, as induced by the renaming scheme of ρ^M.

1. $m_1 \in M$ and $m_2 \in M$. Then $(v, \sigma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v' \cdot \sigma)$ in $b^M(\mathcal{A})$ as well. Notice that in this case $tl(w) = \epsilon$, and thus $w = v$. By definition of α_M (edge in set $\mathcal{P}ub_M(m_1)$) and α'_M, $v \xrightarrow{m_2}_{m_1} v'$ in $\alpha_M(\mathcal{A})$, where $v \models \neg r$, $v_2 \models m_2$ and $v_2 \in E_{\alpha_M(\mathcal{A})}$. Therefore $(v, \gamma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v' \cdot \gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M(v_2, v' \cdot \sigma) = (v_2, v' \cdot \gamma)$.

2. $m_1 \notin M$ and $m_2 \in M$. Then $(v, \sigma) \xrightarrow{m \text{ call } m_2} (v_2, v' \cdot \sigma)$ in $b^M(\mathcal{A})$, where $m = \mathsf{top}^M(v \cdot \sigma)$. By definition of (normal) M-frames $m = \mathsf{top}^M(w)$, and by definition of α_M and α'_M, $w \xrightarrow{m_2}_m v' \cdot tl(w)$ in $\alpha_M(\mathcal{A})$ (since $v \xrightarrow{m_2}_{m_1} v'$ is an edge in $\mathcal{P}ub_M(m_1)$), so that $w \models \neg r$, $v_2 \models m_2$ and $v_2 \in E_{\alpha_M(\mathcal{A})}$. Therefore $(w, \gamma) \xrightarrow{m \text{ call } m_2} (v_2, (v' \cdot tl(w)) \cdot \gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M(v_2, v' \cdot \sigma) = (v_2, (v' \cdot tl(w)) \cdot \gamma)$.

3. $m_2 \notin M$. Then $(v, \sigma) \xrightarrow{\varepsilon} (v_2, v' \cdot \sigma)$ in $b^M(\mathcal{A})$. By definition of α_M and α'_M, $w \rightarrow_{\mathsf{top}^M(w)} \nu(v_2 \cdot v' \cdot tl(w))$ in $\alpha_M(\mathcal{A})$ (first set in ζ) and $w \models \neg r$. Therefore $(w, \gamma) \xrightarrow{\varepsilon} (\nu(v_2 \cdot v' \cdot tl(w)), \gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M(v_2, v' \cdot \sigma) = (\nu(v_2 \cdot v' \cdot tl(w)), \gamma)$.

(return) Let $v \models m_2 \wedge r$, $\sigma \neq \epsilon$ and $hd(\sigma) \models m_1$. Then $(v, \sigma) \xrightarrow{m_2 \text{ ret } m_1} (hd(\sigma), tl(\sigma))$ in $b(\mathcal{A})$. Again, we consider three cases, as induced by the renaming scheme of ρ^M.

1. $m_1 \in M$ and $m_2 \in M$. Then $(v, \sigma) \xrightarrow{m_2 \text{ ret } m_1} (hd(\sigma), tl(\sigma))$ in $b^M(\mathcal{A})$ as well. Notice that in this case $tl(w) = \epsilon$, thus $w = v$, $\gamma = \beta_M(\sigma)$, $\gamma \neq \epsilon$, and $hd(hd(\gamma)) \models m_1$. Therefore $(v, \gamma) \xrightarrow{m_2 \text{ ret } m_1} (hd(\gamma), tl(\gamma))$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M(hd(\sigma), tl(\sigma)) = (hd(\gamma), tl(\gamma))$.

2. $m_1 \notin M$ and $m_2 \in M$. Then $(v, \sigma) \xrightarrow{m_2 \text{ ret } m} (hd(\sigma), tl(\sigma))$ in $b^M(\mathcal{A})$, where $m = \mathsf{top}^M(\sigma)$. Also in this case $tl(w) = \epsilon$, thus $w = v$, $\gamma = \beta_M(\sigma)$, $\gamma \neq \epsilon$, and $hd(hd(\gamma)) \models m_1$. By definition of α_M and α'_M, $m = \mathsf{top}^M(hd(\gamma))$; therefore $(v, \gamma) \xrightarrow{m_2 \text{ ret } m} (hd(\gamma), tl(\gamma))$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M(hd(\sigma), tl(\sigma)) = (hd(\gamma), tl(\gamma))$.

3. $m_2 \notin M$. Then $(v,\sigma) \xrightarrow{\varepsilon} (hd(\sigma), tl(\sigma))$ in $b^M(\mathcal{A})$. We make a case distinction on whether $hd(tl(w)) = hd(\sigma)$, *i.e.* whether the edge that we use to simulate the return was created for a non-recursive call, or not.

- Case $hd(tl(w)) = hd(\sigma)$. By definition of α_M and α'_M, $w \rightarrow_{\mathsf{top}^M(w)} tl(w)$ in $\alpha_M(\mathcal{A})$, and $w \models \neg r$. Therefore $(w,\gamma) \xrightarrow{\varepsilon} (tl(w),\gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M\,(hd(\sigma), tl(\sigma)) = (tl(w),\gamma)$.
- Otherwise there must be a node w' in $\alpha_M(\mathcal{A})$ such that $\nu(v \cdot w') = w$ and $hd(\sigma) = hd(w')$. By definition of α_M and α'_M there must be an edge $w \rightarrow_{\mathsf{top}^M(w)} w'$ in $\alpha_M(\mathcal{A})$, and $w \models \neg r$. Therefore $(w,\gamma) \xrightarrow{\varepsilon} (\sigma' \cdot tl(w),\gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α'_M, $\alpha'_M\,(hd(\sigma), tl(\sigma)) = (\sigma' \cdot tl(w),\gamma)$.

This concludes the proof. □

Notice that in general we do not have behavioural simulation equivalence. The inlining construction introduces transfer edges for calls to and returns from private methods. Because of the latter, the behaviour of the inlined applet can contain a silent transition corresponding to a return from a private method (in the original applet), even when the inlined applet has not yet followed a silent transition corresponding to a call to this private method (in the original applet). The inlining thus introduces new behaviours. Notice however, that these new behaviours are only observable in applets which are not last-call recursive.

A set of methods is *recursive* if every method in the set contains a (reachable) call edge to some method in the set. A call edge is recursive if the calling and the called methods belong to some minimal (and thus, mutually) recursive method set. A program is called *last-call recursive* if from any destination node of any recursive call edge, only transfer edges are reachable. In addition, we shall assume that a return node is reachable from every such destination node.

For last-call recursive applets, we prove the reverse correspondence for observable behaviours.

Theorem 4. *Let $\mathcal{A} : I$ be a closed last-call recursive applet, and let $M \subseteq I^+$. Then $b(\alpha_M(\mathcal{A})) \leq_w b^M(\mathcal{A})$.*

Proof. Consider a state (w,γ) in $b(\alpha_M(\mathcal{A}))$, where $\lambda_{\mathsf{Meth}}(hd(w)) \notin M$ and $hd(w) \models r$. For last-call recursive applets, the inlining transformation α_M has the property that for any such w, the nodes w' such that $\nu(hd(w) \cdot w') = w$ but $hd(w) \cdot w' \neq w$, and which are structurally reachable from w in $\alpha_M(\mathcal{A})$ form (together with w) a strongly connected component and are equivalent *w.r.t.* structural simulation. As a consequence, in $b(\alpha_M(\mathcal{A}))$, all states (w',γ) for a given γ also form a strongly connected component and are weak simulation equivalent. Modulo such "return" equivalence classes, we show by co-induction that $(\alpha'_M)^{-1}$ is a weak simulation between $b(\alpha_M(\mathcal{A}))$ and $b^M(\mathcal{A})$. More exactly, we show that (1) the valuations of $\alpha'_M(v,\sigma)$ and (v,σ) agree, and (2) if $\alpha'_M(v,\sigma) \xrightarrow{l} (w',\gamma')$ is a transition in $b(\alpha_M(\mathcal{A}))$ other than a (silent) transition within a return equivalence class, then $(v,\sigma) \stackrel{l}{\Rightarrow} (v',\sigma')$ in

$b^M(\mathcal{A})$ for some $v'$and σ' such that $\alpha'_M(v',\sigma') = (w',\gamma')$ (in most cases we even show the corresponding strong transition). The result then follows since α'_M maps entry states of $b(\alpha_M(\mathcal{A}))$ to entry states of $b^M(\mathcal{A})$.

Let (v,σ) be a configuration of $b^M(\mathcal{A})$, and let $\alpha'_M(v,\sigma) = (w,\gamma)$. Note that $hd(w) = v$, and that χ is invoked with arguments $\lambda_{\mathsf{Meth}}(hd(w))$ and $tl(w)$ in the construction of $\alpha_M(\mathcal{A})$. It is easy to check that the valuations agree, so we focus on the transitions. We consider the different cases leading to transitions from configuration (w,γ) in $b(\alpha_M(\mathcal{A}))$, as induced by the transition rules for closed applets given in Table 1.

(transfer) Let $m \in M$, $w \rightarrow_m w'$ and $w \models \neg r$. Then $(w,\gamma) \xrightarrow{\varepsilon}(w',\gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α_M, there are three possible cases for the transfer edge $w \rightarrow_m w'$ to appear in $\alpha_M(\mathcal{A})$, which we consider in turn.

1. $w' = v' \cdot tl(w)$ and $v \rightarrow_{m'} v'$ for some v' and $m' = \lambda_{\mathsf{Meth}}(v) = \lambda_{\mathsf{Meth}}(v')$, and $v \models \neg r$. Then $(v,\sigma) \xrightarrow{\varepsilon}(v',\sigma)$ in $b(\mathcal{A})$, and hence also in $b^M(\mathcal{A})$. By the definition of α'_M, $\alpha'_M(v',\sigma) = (w',\gamma)$.

2. (internal call) $w' = \nu(e \cdot v' \cdot tl(w))$, $e \models m'$ and $e \in E$, $m' \notin M$, there is a call edge $v \xrightarrow{m'}_{m''} v'$ in $\mathcal{A}$ for some m'', $v \models \neg r$. Then $(v,\sigma) \xrightarrow{m'' \text{ call } m'} (e, v' \cdot \sigma)$ in $b(\mathcal{A})$, and hence $(v,\sigma) \xrightarrow{\varepsilon} (e, v' \cdot \sigma)$ in $b^M(\mathcal{A})$. By definition of α'_M, $\alpha'_M(e, v' \cdot \sigma) = (w',\gamma)$.

3. (internal return) $w = \nu(v \cdot w')$, $v \models m' \wedge r$, $m' \notin M$ and there is a call edge $v' \xrightarrow{m'}_{m''} v''$ in $\mathcal{A}$ for $v'' = hd(w')$ and some m'' and v' such that $v' \models \neg r$. We consider three subcases.

   - $hd(\sigma) = v''$. Then $(v,\sigma) \xrightarrow{m' \text{ ret } m''} (hd(\sigma), tl(\sigma))$ in $b(\mathcal{A})$, and hence $(v,\sigma) \xrightarrow{\varepsilon} (hd(\sigma), tl(\sigma))$ in $b^M(\mathcal{A})$. By tdefinition of α'_M, $\alpha'_M(hd(\sigma), tl(\sigma)) = (w',\gamma)$.
   - $hd(\sigma) \neq v''$ and $w = v \cdot w'$. Then we are dealing with a return from a recursive call in $b(\mathcal{A})$, and there must be a decomposition $\sigma' \cdot v'' \cdot \sigma''$ of σ such that $\lambda_{\mathsf{Meth}}(\sigma_0) = m'$ and no node of σ' is in M. Since $\mathcal{A}$ is last-call recursive, all nodes in σ' are either return nodes or nodes leading to return nodes via transfer paths only. Therefore $(v,\sigma) \stackrel{\varepsilon}{\Rightarrow} (v'',\sigma'')$ in $b^M(\mathcal{A})$. By the definition of α'_M, $\alpha'_M(v'',\sigma'') = (w',\gamma)$.
   - $hd(\sigma) \neq v''$ and $w \neq v \cdot w'$. Then (w,γ) and (w',γ) are in the same return equivalence class (see above), so we do not have to consider this case.

(call) Let $m_1, m_2 \in M$, $w \xrightarrow{m_2}_{m_1} w'$, $w \models \neg r$, $w'' \models m_2$ and $w'' \in E$. Then $(w,\gamma) \xrightarrow{m_1 \text{ call } m_2} (w'', w' \cdot \gamma)$ in $b(\alpha_M(\mathcal{A}))$. By definition of α_M, we must have $w' = v' \cdot tl(w)$ for some v' such that, in $\mathcal{A}$, $v \xrightarrow{m_2}_m v'$ and $v \models \neg r$ for some private method m of m_1. By definition of α'_M, $w'' = v''$ for some v'' such that $v'' \models m_2$ and $v'' \in E$. Then $(v,\sigma) \xrightarrow{m \text{ call } m_2} (v'', v' \cdot \sigma)$ in $b(\mathcal{A})$, and since by definition of α'_M, $\mathsf{top}^M(v \cdot \sigma) = m_1$, we have $(v,\sigma) \xrightarrow{m_1 \text{ call } m_2} (v'', v' \cdot \sigma)$ in $b^M(\mathcal{A})$. By definition of α'_M, $\alpha'_M(v'', v' \cdot \sigma) = (w'', w' \cdot \gamma)$.

(return) Let $m_1, m_2 \in M$, $w \models m_2 \wedge r$, $\gamma \neq \epsilon$ and $hd(\gamma) \models m_1$. Then $(w,\gamma) \xrightarrow{m_2 \text{ ret } m_1} (hd(\gamma), tl(\gamma))$ in $b^M(\alpha_M(\mathcal{A}))$. By definition of α_M and α'_M, we must have $w = v$ and $v \models m_2 \wedge r$. Also, $\sigma \neq \epsilon$ and hence $(v,\sigma) \xrightarrow{m_2 \text{ ret } m} (hd(\sigma), tl(\sigma))$ in $b(\mathcal{A})$ for $m = \lambda_{\mathsf{Meth}}(hd(\sigma))$. By definition of α'_M, $\mathsf{top}^M(\sigma) = m_1$ and hence $(v,\sigma) \xrightarrow{m_2 \text{ ret } m_1} (hd(\sigma), tl(\sigma))$ in $b^M(\mathcal{A})$. By definition of α'_M, $\alpha'_M(hd(\sigma), tl(\sigma)) = (hd(\gamma), tl(\gamma))$.

This concludes the proof. □

Since weak simulation contains simulation we have the following.

Corollary 1. *Let $\mathcal{A} : I$ be a closed last-call recursive applet, and let $M \subseteq I^+$. Then $b^M(\mathcal{A}) \equiv_w b(\alpha_M(\mathcal{A}))$.*

6 Interface Abstraction and Compositional Reasoning

Using the results obtained above, we can state several verification principles that can be used to prove properties of applet interface behaviour. We first present two abstraction principles, and then we show how these can be combined with our compositional verification principle (from [9]).

Interface Abstraction. Let $\mathcal{A} : I$ be a closed applet, and let $M \subseteq I^+$. With the results established above, we can justify the following abstraction principle, where ψ is a behavioural interface formula.

$$(\mathsf{abstract})\ \frac{\alpha_M(\mathcal{A}) \models_b \psi}{\mathcal{A} \models_b^M \psi}$$

Theorem 5. *Rule* (abstract) *is sound.*

Proof. Follows from the definition of behavioural satisfaction, Theorem 3, Theorem 1, and the definition of behavioural interface satisfaction. □

When $\mathcal{A}$ has last-call recursion, we can even provide a faithful abstraction principle for properties of the observable behaviour by using transformation δ from Section 2.

$$(\mathsf{weak\text{-}abstract})\ \frac{\alpha_M(\mathcal{A}) \models_b \delta(\psi)}{\mathcal{A} \models_{b,w}^M \psi}$$

Theorem 6. *Rule* (weak-abstract) *is sound and complete.*

Proof. Follows from the definition of behavioural satisfaction, Proposition 1, Corollary 1, Theorem 2, and the definition of weak behavioural interface satisfaction, all of which are equivalences. □

Compositional Reasoning. In earlier work [9] we presented the following sound and complete compositional verification principle:

$$(\mathsf{compos})\ \frac{\mathcal{A} \models_s \sigma \qquad \mathcal{M}ax_{I_\mathcal{A}}(\sigma) \uplus \mathcal{B} \models_b \psi}{\mathcal{A} \uplus \mathcal{B} \models_b \psi}\ \mathcal{A} : I_\mathcal{A}$$

Here $\mathcal{A}$ and $\mathcal{B}$ are applets, such that $\mathcal{A} \uplus \mathcal{B}$ is a closed applet. Further, σ is a formula in simulation logic on the structural level (*i.e.* boxes are interpreted over the edges in the call graph), while ψ is a property at the behavioural level[2]. Finally, $Max_{I_\mathcal{A}}(\sigma)$ is a construction described in [9] yielding a so-called *maximal applet w.r.t.* σ and $I_\mathcal{A}$, *i.e.* an applet with interface $I_\mathcal{A}$ that simulates all other applets with this interface satisfying property σ. We combine this rule with the abstraction principle above to obtain the following abstract compositional verification principle:

$$(\mathsf{abstract\text{-}compos})\ \frac{\alpha_M(\mathcal{A}) \models_s \sigma \qquad \mathcal{M}ax_{I_{\alpha_M(\mathcal{A})}}(\sigma) \uplus \mathcal{B} \models_b \psi}{\mathcal{A} \uplus \mathcal{B} \models_b^{M \cup I^+_\mathcal{B}} \psi} \quad \begin{array}{l}\mathcal{A} : I_\mathcal{A},\\ I^-_\mathcal{B} - I^+_\mathcal{B} \subseteq M\end{array}$$

Theorem 7. *Rule* (abstract-compos) *is sound.*

Proof. Follows from the abstraction and the compositional verification principle, plus Propositions 4 and 5. □

Notice that the interface of required methods that is used for the maximal model construction uses $I^-_\mathcal{A} - I^+_\mathcal{A}$. Typically, this will correspond to the public interface of $\mathcal{B}$, and for each implementation of $\mathcal{A}$ it should be checked whether it respects this public interface of $\mathcal{B}$.

Finally, similarly as for the abstraction principle, we can state a faithful compositional verification principle for properties of the observable interface behaviour of applets which are last-call recursive.

$$(\mathsf{weak\text{-}abstract\text{-}compos})\ \frac{\alpha_M(\mathcal{A}) \models_s \sigma \quad \mathcal{M}ax_{I_{\alpha_M(\mathcal{A})}}(\sigma) \uplus \mathcal{B} \models_b \delta(\psi)}{\mathcal{A} \uplus \mathcal{B} \models_{b,w}^{M \cup I^+_\mathcal{B}} \psi} \quad \begin{array}{l}\mathcal{A} : I_\mathcal{A},\\ I^-_\mathcal{B} - I^+_\mathcal{B} \subseteq M\end{array}$$

Theorem 8. *Rule* (weak-abstract-compos) *is sound and complete.*

7 Practical Impact of Inlining

As explained above, we are interested in studying the abstract behaviour of applets, because in a truly compositional setting nothing is known about the different components,

[2] A similar principle exists if ψ is a structural property, since behavioural simulation contains structural simulation.

except (some properties of) their interface behaviour. For a newly downloaded applet we only require that it implements the shareable interface; we do not put any restrictions on *how* it implements this shareable interface, except that the implementation should respect the global security requirements. Studying compositional verification at the abstract level allows to specify the local and global properties at the abstract level, without taking any implementation details into account. Moreover, when considering shareable interfaces only, the maximal models that we compute to verify the decomposition of the global property into the local ones are significantly reduced in size, making the verification much more efficient.

In order to show the impact of abstraction and inlining on a realistic case study, this section revisits the electronic purse case study [5], specifying an illicit interaction between applets *Purse* and *Loyalty*. In the original case study we computed maximal applets using the implementation interfaces (containing about 300 methods per applet). This was time-consuming (25 mins. to 13 hrs.) and moreover, the size of the outcome was so large that verification was unfeasible. However, the public interfaces (*i.e.* the shareable interfaces) of these applets both provide only 4 methods. If we refer to the shareable interfaces as SI_P (methods provided by *Purse* for *Purse* and *Loyalty*) and SI_L(*Loyalty* for *Purse* and *Loyalty*), respectively, we can identify the following public interfaces: $(SI_P, SI_P \cup SI_L)$ for *Purse*, and $(SI_L, SI_P \cup SI_L)$ for *Loyalty*.

We use the tool set described in [5], plus an implementation of the inlining algorithm in Ocaml to redo the case study at the abstract level. For convenience we repeat the global and local specifications, but this time specified at the interface level; for further motivations we refer to [5].

The global specification ψ says that a call to *Loyalty.logFull* does not trigger any calls to any other loyalty, including indirect communications, via the *Purse*. The specification uses several abbreviations for readability (where $\mathcal{A}$ is an applet such that $\mathcal{A} : (I^+, I^-)$ and M a set of methods).

$$\begin{array}{rcl} \mathit{Always}\,\phi & = & \nu Z.\,\phi \wedge [L_b]Z \\ \mathit{Within}\,m\,\phi & = & \neg m \vee (\mathit{Always}\,\phi) \\ \mathit{CanNotCall}\,\mathcal{A}\,M & = & \bigwedge_{m \in I^+} \bigwedge_{m' \in M} [m\,\mathit{call}\,m']\,\mathsf{false} \end{array}$$

$$(\psi)\ \mathit{Within\ Loyalty.logFull}\ (\mathit{CanNotCall\ Loyalty}\ SI_L) \wedge (\mathit{CanNotCall\ Purse}\ SI_L)$$

For the *Loyalty* applet we exclude any external calls, except those to the methods *Purse.isThereTransaction* and *Purse.getTransaction* (σ_L). For the *Purse* applet we specify that both these methods do not make any external calls (σ_P). Again we use several abbreviations.

$$\begin{array}{rcl} \mathit{Everywhere}\,\sigma & = & \nu Z.\,\sigma \wedge [\varepsilon, I^-]Z \\ M\ \mathit{HasNoCallsTo}\ M' & = & \left(\bigwedge_{m \in M} \neg m\right) \vee (\mathit{Everywhere}\ [M']\,\mathsf{false}) \\ \mathit{HasNoOutsideCalls}\,M & = & M\ \mathit{HasNoCallsTo}\ (I^- \setminus M) \end{array}$$

	$\mathcal{M}ax(\sigma_L)$	$\mathcal{M}ax(\sigma_L)$ in [5]	$\mathcal{M}ax(\sigma_P)$	$\mathcal{M}ax(\sigma_P)$ in [5]
#nodes	8	474	8	2786
#edges	120	277 700	88	603 128
constr. time	0.05 s.	25 min.	0.05 s.	13 hrs.

Table 2: Size and timing for maximal model construction.

$$
\begin{array}{ll}
(\sigma_L) & \mathit{loyalty.logFull}\ \mathit{HasNoCallsTo} \\
 & \qquad (SI_P \cup SI_L)/\{\ \mathit{Purse.isThereTransaction}, \\
 & \qquad\qquad \mathit{Purse.getTransaction}\} \\
(\sigma_P) & \mathit{HasNoOutsideCalls}\ \mathit{Purse.isThereTransaction} \wedge \\
 & \mathit{HasNoOutsideCalls}\ \mathit{Purse.getTransaction}
\end{array}
$$

To redo the case study at the abstract level, we take the following steps (where P and L denote implementations of *Purse* and *Loyalty*, respectively):

- compute $\mathcal{M}ax_{(SI_P, SI_P \cup SI_L)}(\sigma_P)$ and $\mathcal{M}ax_{(SI_L, SI_L \cup SI_P)}(\sigma_L)$ using the Maximal Model constructor [9, 5];
- model check $\mathcal{M}ax_{(SI_P, SI_P \cup SI_L)}(\sigma_P) \uplus \mathcal{M}ax_{(SI_L, SI_L \cup SI_P)}(\sigma_L) \models_b \delta(\psi)$ using a prototype implementation of a model checker for PDAs;
- compute $\alpha_{SI_P}(P)$ and $\alpha_{SI_L}(L)$ using the inlining algorithm; and
- model check $\alpha_{SI_P}(P) \models_s \sigma_P$ and $\alpha_{SI_L}(L) \models_s \sigma_L$ using CWB [4].

Table 2 compares the outcome and timing for the maximal model construction with the corresponding step in the original case study. Checking the correctness of the decomposition took approximately 5 seconds. The inlining algorithm took 0.6 seconds on both *Loyalty* and *Purse*. Even though theoretically the worst-case blowup in the number of nodes of the inlined applets, determined by the number of normal M-frames, is exponential in the number of private methods, in practice this is not likely to happen. In our case we even observed a reduction in size of the graphs, due by the fact that the inlining focuses on interaction with other applets, and thus any code that is executed only when the applet is selected and receives commands from the runtime environment, is left out by the inlining. Verifying the local property on the inlined applets of *Loyalty* and *Purse* took approximately 15 and 10 seconds, respectively.

8 Conclusions

When providing a set of functionalities to the outside world, it is nowadays common practice to use encapsulation to hide internal implementation details. One encapsulation mechanism

is the partitioning of procedures into public and private ones. However, program verification techniques often do not handle encapsulation. Therefore we propose a notion of interface behaviour, to describe the behaviour of programs, abstracting from the internal, private behaviour. We also propose a program transformation based on inlining of private methods, for which we prove that it preserves the interface behaviour. Since the interface behaviour of an inlined program coincides with its behaviour, the inlining transformation provides a means for checking properties of the interface behaviour of the original program by reduction to a standard verification problem. Such a focus on interface behaviour is significant from a methodological, software engineering point of view, and it even becomes paramount in any truly compositional setting, where verification has to be based on assumptions about the interface behaviour of not yet available components (see [5, 9]).

The inlining transformation presented in this paper is an interface behaviour preserving abstraction. It reduces the number of methods of a program to the (in practice much smaller) number of its public methods. This allows to specify the behaviour of components in an abstract and more natural way, without any dependence on implementation details. Moreover, this has a crucial impact on the practical applicability of the maximal model construction from [9], which is an essential ingredient of our compositional verification technique, and which is exponential in the number of methods. By applying interface abstraction, we are now able to construct in fractions of a second what otherwise is on the limits of the capacity of modern computers.

Acknowledgements

We thank Gennady Chugunov for helping us redoing the verifications.

References

[1] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Analysis and Construction of Software, TACAS 04*, number 2998 in LNCS, pages 467–481. Springer, 2004.

[2] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.

[3] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[4] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proc. 9th IFIP Symp. Protocol Specification, Verification and Testing*, 1989.

[5] M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, FASE 2004*, number 2984 in LNCS, pages 84–98. Springer, 2004.

[6] O. Kaser and C.R. Ramakrishnan. Evaluating inlining techniques. *Journal of Computer Languages (JCL)*, 24(2):55–72, 1998.

[7] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.

[8] C. Sprenger, D. Gurov, and M. Huisman. Simulation logic, applets and compositional verification. Technical Report RR-4890, INRIA, 2003.

[9] C. Sprenger, D. Gurov, and M. Huisman. Compositional verification for secure loading of smart card applets. In *Formal Methods and Models for Co-Design (Memocode 2004)*, pages 211–222. IEEE, 2004.

[10] C. Stirling. *Modal and Temporal Logics of Processes.* Springer, 2001.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399